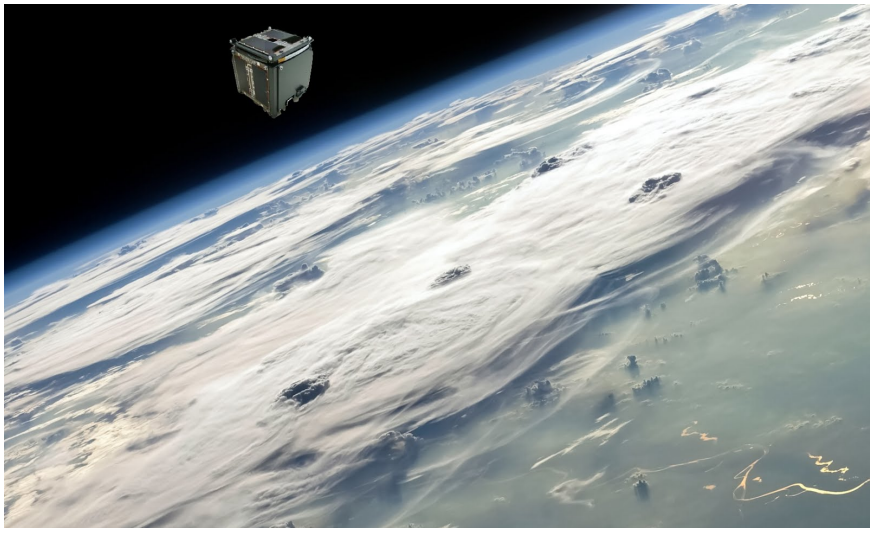
Implementing a MATLAB Based Attitude Determination Algorithm in C within the PolySat Software Architecture

By Dominic Bertolino

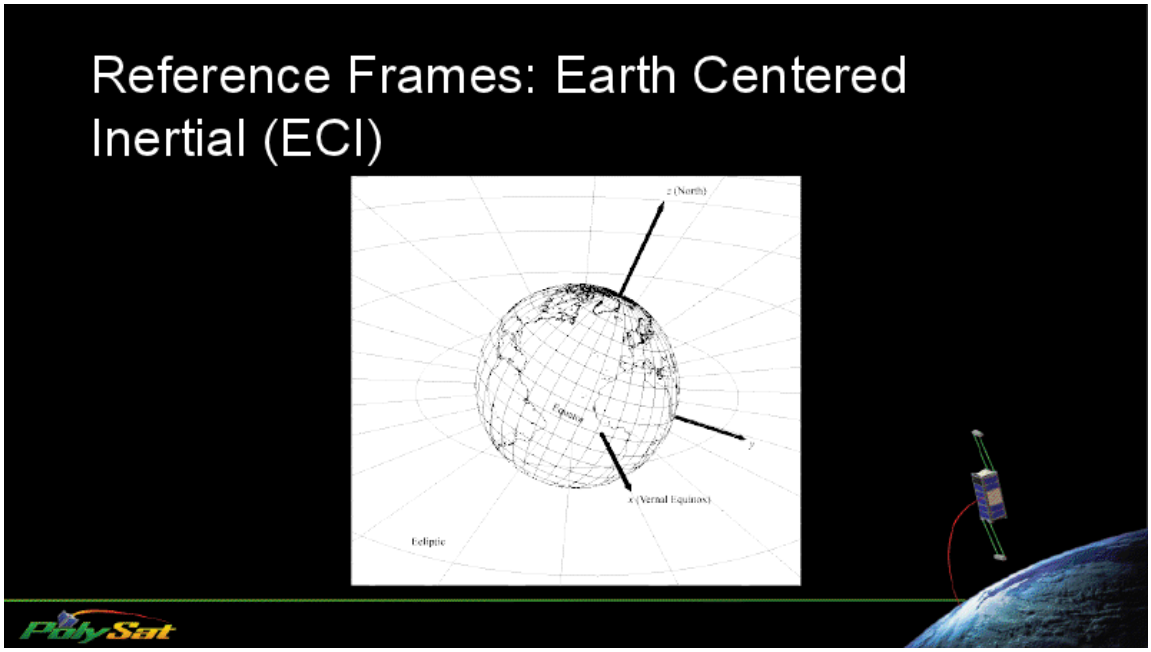


Reference

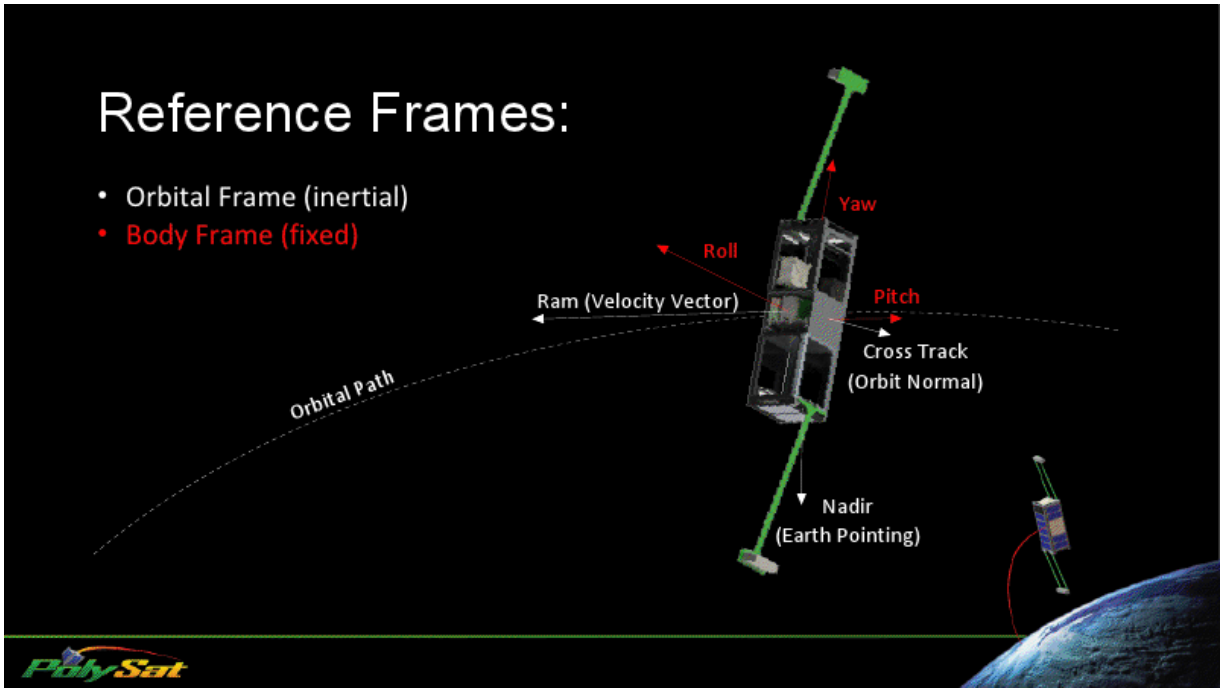
The various acronyms and standards used throughout this paper are first produced here for reference.

Coordinate Systems

Earth Centered Initial:
This coordinate system is centered on the earth and rotates with it.



Body Framed Fixed (BFF) and Orbital Frame Initial (OFI):
BFF is centered on the satellite and rotates with the satellite. OFI is also centered on the satellite but is always facing the direction of movement (i.e., the orbit).



Acronyms

ADCS - Attitude Determination and Control System
CS - Coordinate System
BFF - Body Frame Fixed
ECI - Earth Centered Inertial

Overview

This project focuses on one component within a complete attitude determination and control system (ADCS) for a small satellite. The component consists of porting the algorithm that determines the current attitude of the satellite developed by AERO students / team members. The original algorithm has been developed in MATLAB code. The actual algorithm will be simulated and tested in MATLAB by the AEROs. The porting consisted of integrating the pieces into the custom PolySat software environment in C. Testing was done to verify the ported component corresponded to the original MATLAB component as well as verify its runtime on the PolySat embedded system was acceptable.

It will be useful to detail the entire ADCS along with the current PolySat software architecture as this is the context that ultimately determines the design of the determination algorithm and the form its implementation takes within the PolySat software environment. After an overview, a more detailed look at the determination algorithm with different approaches and the chosen design will be covered.

ADCS Architecture

The major components of the ADCS system that will be covered in the overview include all the components for both the determination and control portions of the ADCS. Below is a picture showing the different components and their relationships with each other in terms of data flow. Each component is explained below.

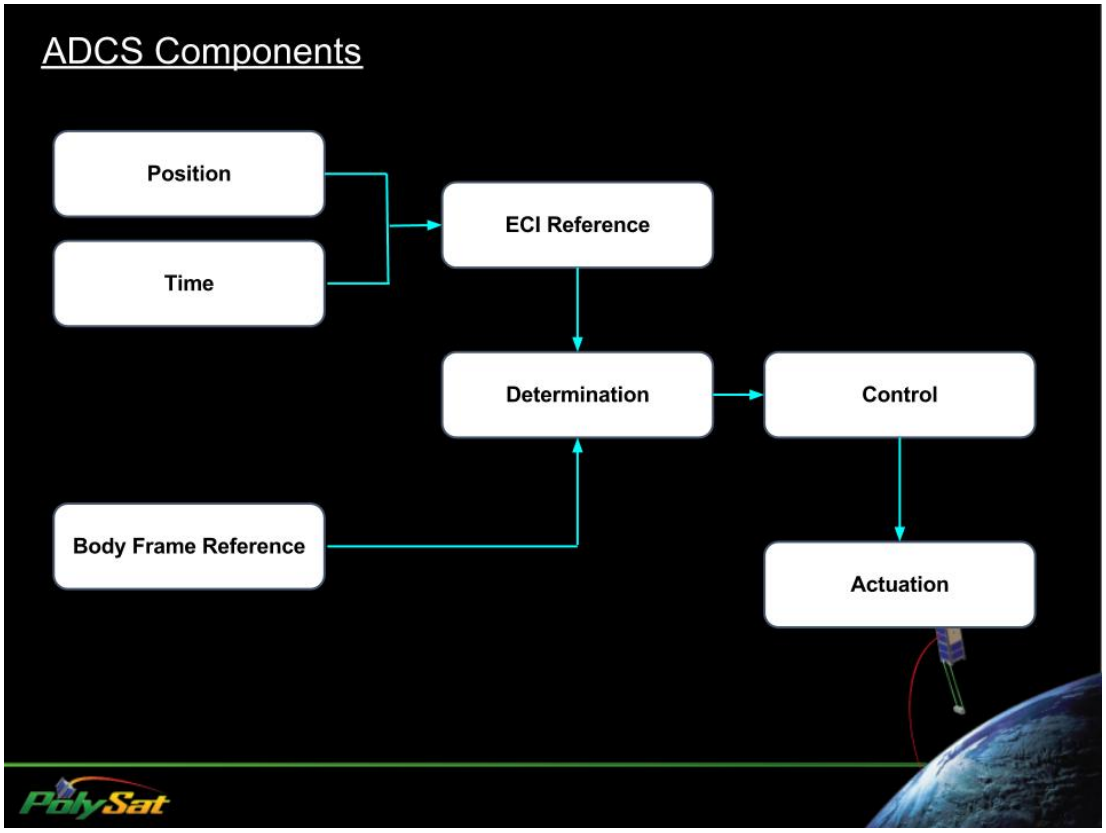


Figure 1: ADCS Components

Determination

There are many components that go into the task of determining the attitude of the satellite. The end goal of this system will be to have knowledge of the attitude of the satellite in earth centered coordinates (ECI). This knowledge will ultimately take the form of a quaternion. From this, the control system will take this quaternion along with one giving the desired attitude and produce the necessary torques to adjust the satellite. Following is a description of the major components of the determination subsystem.

Body Frame Reference

In this subsystem, data from the two primary types of sensors, magnetometers and sun sensors, will be gathered. These sensors will produce a reference vector.

For each sensor type, three readings corresponding to the three components of a 3D vector will be gathered. This vector can be related to the defined body frame fixed coordinate system (BFF-CS) of the satellite via sensor mounting info. This is the primary goal of this particular software component. This vector will ultimately be combined with a ECI vector reading of the same natural reference (sun or magnetic field) and thereby give us a way to relate the attitude of the satellite to a standard reference frame, ECI, allowing desired attitudes to be compared with the measured ones creating a means of acquiring an attitude delta.

Position

The satellite position is needed in order to get an accurate ECI reference vector. An orbital propagator allows us to know the exact position of our satellite. A position will initially be uploaded at a certain time. Afterwards, the orbital propagator will be able to give us the position based solely on the time that has passed. For this to be accurate, the time must be in sync taking additional software to counter clock drift. The ‘time’ component is referring to this syncing software.

ECI Reference

The two natural reference phenomenons being used, the sun and earth’s magnetic field will need to be known in relation to both the satellite and the earth (via vectors) or, as mentioned, in the BFF-CS and ECI-CS. Both these phenomena are well documented and standard databases exist giving the actual value for any position our satellite will be in during its orbit. Using this fact, the ECI-CS reference vector can be acquired. This is not to be confused with the ECI-CS vector that is the end goal. Two components are needed for each vector type: the satellite’s position and the database for the corresponding phenomenon. The ‘ECI Reference’ software component will take in the position of the satellite and combine them with the corresponding databases mentioned below to produce the ECI-CS reference vector.

Earth’s magnetic field model: The model that will be used is the World Magnetic Model produced by the National Geophysical Data Center. This is a well known, standard database giving a mathematical description of earth’s magnetic field allowing one to calculate the value for any position. This model is a standard for such government organizations as the DoD [2]. The issue is onboard calculations are too expensive leading to the need for an additional software component composed of precalculated values for the known orbit of the satellite that can be efficiently stored and retrieved. The form this component will take is a look-up table that does 3-D interpolation.

Sun ephemeris**:** The position of the sun will be acquired using the sun ephemeris gathered by the astronomical almanac.

Determination Algorithm

The primary purpose of this component is to produce a quaternion. From previous components, a vector in ECI can be acquired by combining the ECI reference vector with the BFF vector. However, this does not give the rotation effectively giving an infinite amount of possible attitudes. In order to determine the rotation, a value that can't be measured, we will use measurements over time, a dynamics model describing the movement of the satellite, and a Kalman filter to produce a quaternion which includes the rotation. The main focus of this senior project has been coordinating with the AEROs to implement this component in the PolySat software environment and is therefore elaborated on later in this report.

Control

The control component will take in two ECI quaternions containing attitude information: the measured quaternion acquired from the determination component discussed and a quaternion describing the desired attitude. The desired will either be known in advanced for every orbital position or be uploaded from a ground station via the radio. From these two quaternions, the control component will generate a needed torque value.

Actuation

The magnetorquers are the primary way the satellite will control the attitude. Current will be pulse-width-modulated through coils of wire to create different magnitudes of a magnetic field. This field and the actual field will create a torque which will change the attitude of the satellite. The input to this component will be a raw torque value. Software will be developed to translate this torque into the actual hardware control values that will produce the equivalent torque.

PolySat Software Architecture

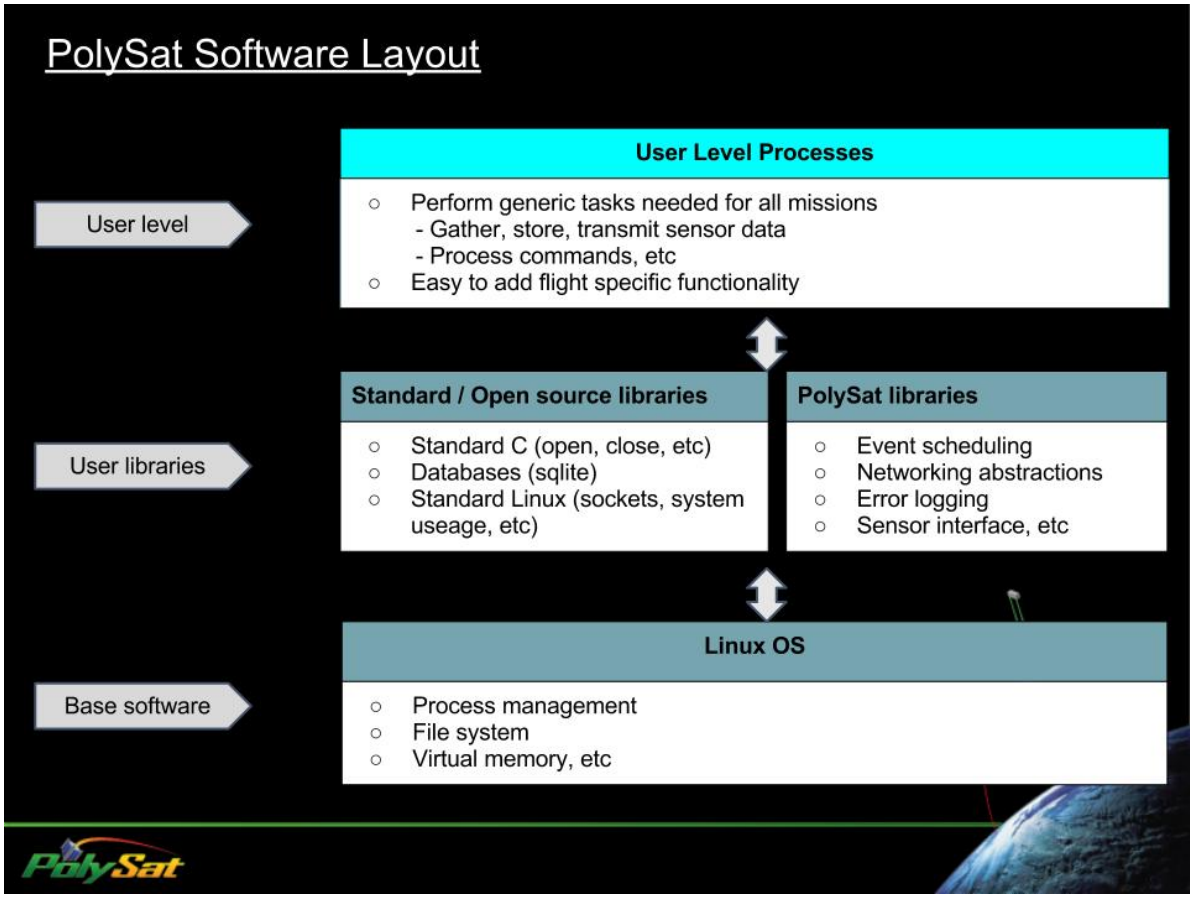


Figure 2: PolySat Software Layout

Base Software

As shown, the PolySat software environment is built upon Linux providing all the convenient OS functionality such as process management, a file system, and virtual memory. Due to the need for efficiency and a small footprint, C is the language of choice. In the case of this senior project, this means that additional functionality, not provided standard through C will need to be either implemented or acquired. Additional libraries already developed by PolySat push towards a unique software environment that emphasizes event driven design and interprocess communication. This framework will be discussed below in the ‘Architecture Design’ section. The ADCS software components will need to integrate with this framework as well as the current user level software that runs for every mission providing basic functionality such as data acquisition, storage, and transmission as well as command handling.

Architecture Design

Major software components are broken up into processes, taking advantage of the process abstraction provided by the Linux OS to allow for modular development and memory isolation. In figure 3 below, the most essential PolySat processes are shown on the left. Their major responsibilities is mentioned in the figure and includes data acquisition, storage, and transmission. Commonly on a satellite, events need to happen at a precise time making event driven design an ideal approach. The PolySat libraries provide this framework and all processes are designed to utilize this functionality. To briefly describe this framework, processes can schedule time based events or event based events. Time based events are triggered after a certain set time while event based events trigger on the activity of a watched file descriptor. This functionality gives the Linux OS RTOS like functionality. Along with event driven design, processes utilize further PolySat libraries to exchange data through interprocess communication. Processes exchange data with one another using the UDP protocol utilizing the standard Linux networking libraries with additional PolySat features. This is the environment the ADCS system will be integrated into.

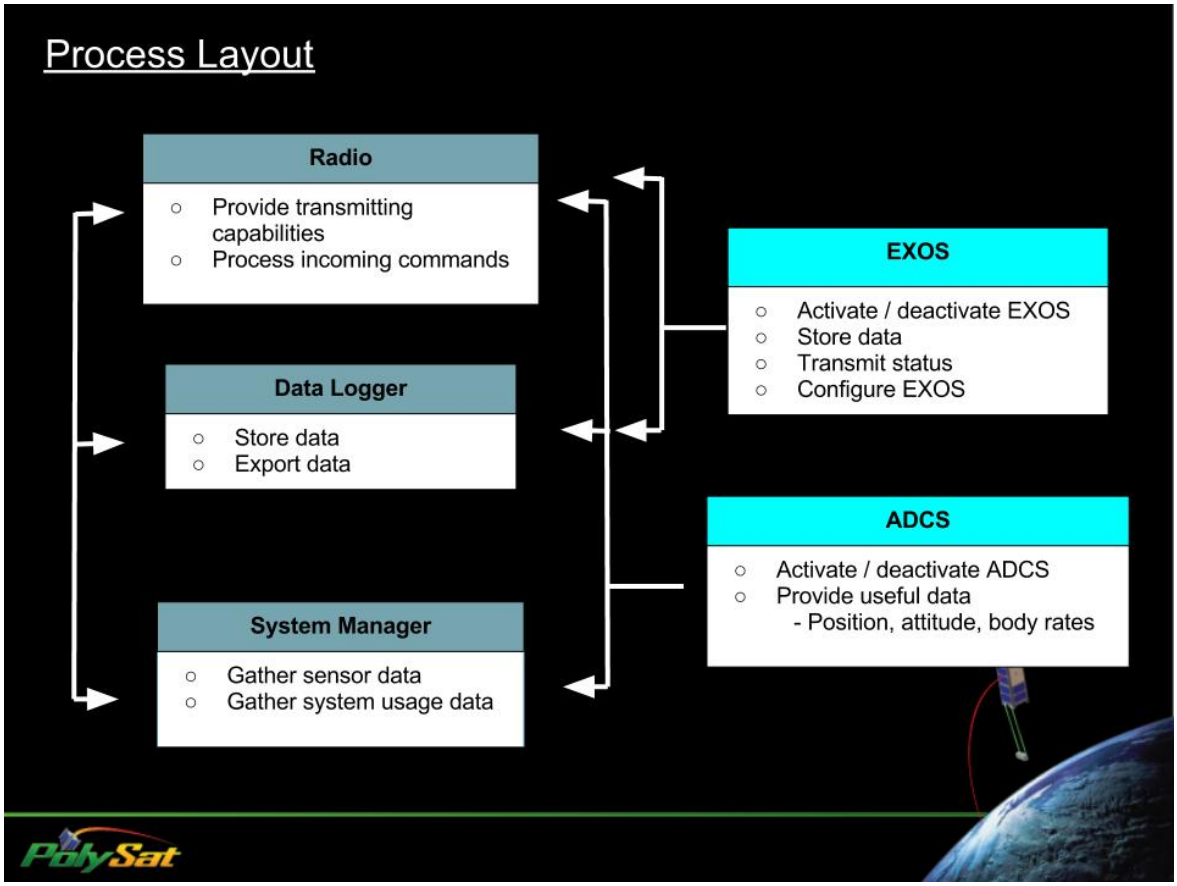


Figure 3: Process Layout

Determination Algorithm Software Component

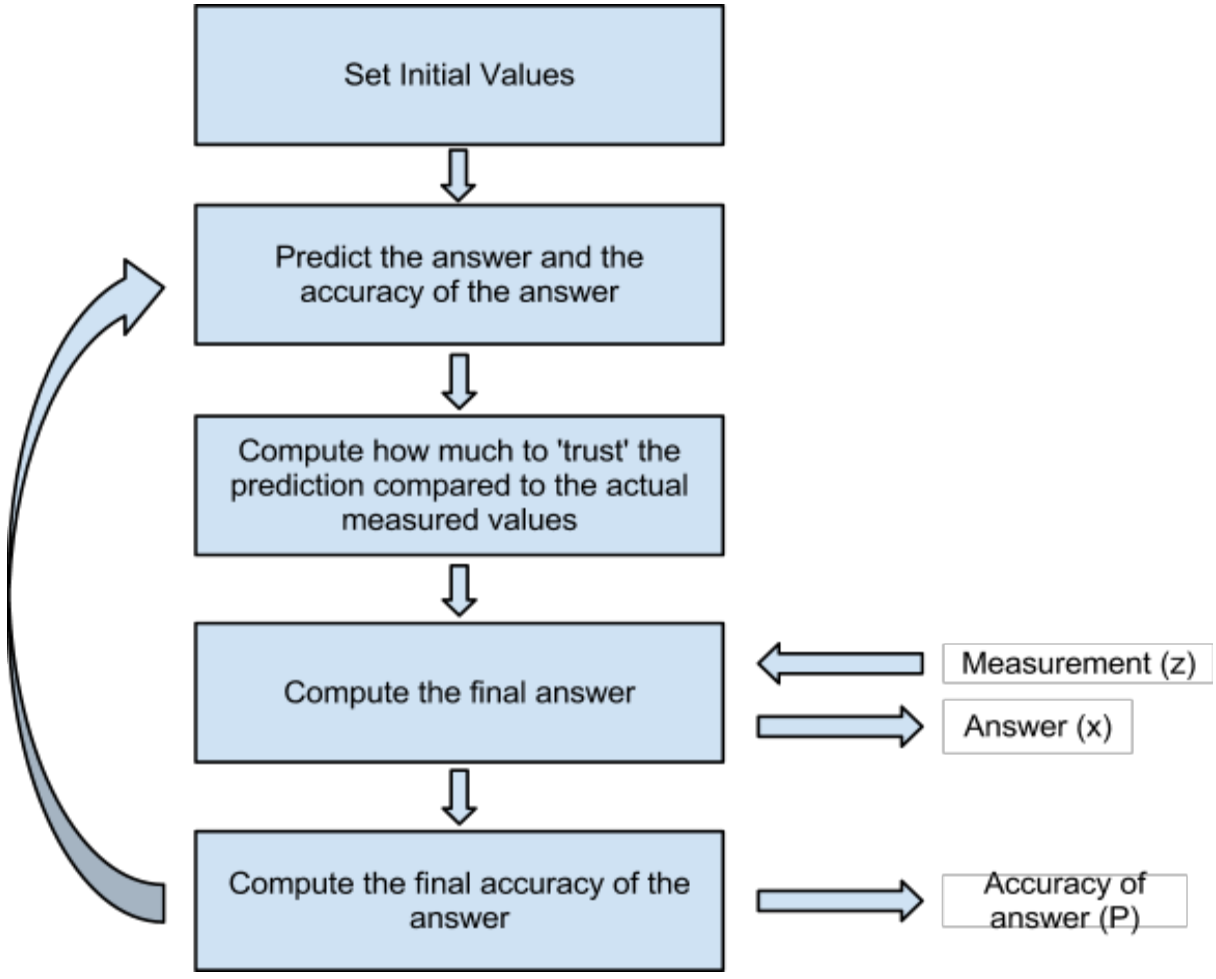
Overview

As mentioned, the primary purpose of the determination algorithm component is to produce a quaternion by combining the ECI reference vector with the BFF vector along with a dynamics model and a Kalman Filter. The actual Kalman filter design is beyond the scope of a CPE and has been handled by AERO students who themselves are heavily looking towards past designs for guidance. With that being said, a basic understanding of the design was still needed to implement the filter in the unique PolySat software environment with a language as stripped down as C.

Kalman Filter Overview

Though the math and derivation behind the Kalman filter is far from trivial, the basic concept is simple and consists of prediction and a feedback loop. The idea is to combine a predicted value (coming from a known dynamics model in this case) and actual measurements to produced an optimized value. The weighting, determining how much to “trust” the prediction vs the measured value, is updated each time and looped back for the next calculation. Below is the typical flow of a Kalman Filter.

Kalman Filter Flow



The following describes the system variables used to create the functionality of the Kalman Filter. These are the variables that can be changed to alter the characteristics of the Kalman Filter. However, they are not typically changed by the filter during runtime with the exception being the state transition matrix which is in certain designs.

System Variables

Phi: The state transition matrix used to calculate the prediction value.

Q: The covariance matrix of the process noise.

R: The covariance matrix for the sensor noise.

H: The observational model matrix that is used to relate the measured value to the predicted

values.

The following describes the variables used internally by the filter. These variables make up the persistent context of the filter and each iteration uses the previous values. To start off the filter, a first guess is needed for some of them. Unlike the state variables, these change during runtime.

Internal Variables
x: The state estimate matrix used as the final output of each filter iteration.
xp: The state prediction matrix used as the intermediate prediction of the upcoming measurement. This is combined with the actual measured values to attain the state estimate, 'x.'
P: The error covariance estimate used to indicate the accuracy of the state estimate.
Pp: The error covariance prediction used in a similar way as the state prediction.
K: The Kalman Gain used to weight how much to use the prediction vs the actual measurement.

For our particular application, we use the predictive aspect to generate a value that was not measured, the rotation. This is not the only benefit of using a Kalman filter though. We will also be able optimize the measurements we are making, namely, the sun and the magnetic field. Optimizing sensor readings is a common use of a Kalman filter and will easy come from our chosen design.

Of the many flavors of Kalman filters, the design chosen takes the form of an multi-state variable, non-linear, extended filter. This design keeps the general idea behind the Kalman filter adding complexities requiring matrix support and partial differentiation. The design was heavily influenced by Todd Humphreys's paper, "Attitude Determination for Small Satellite with Modest Pointing Constraints" [1].

Implementation

Determination Design Overview

As Figure 3 shows, the ADCS component will be designed as its own process. The components of this process dissembled earlier (Figure 1) can be logically thought of as Objects since the design will leverage C to be Object like. Therefore, implementing the determination algorithm component, the main focus of this project, will be similar to implementing an Object within a PolySat process utilizing event driven design and interprocess communication. This Object is described in Figure 4.

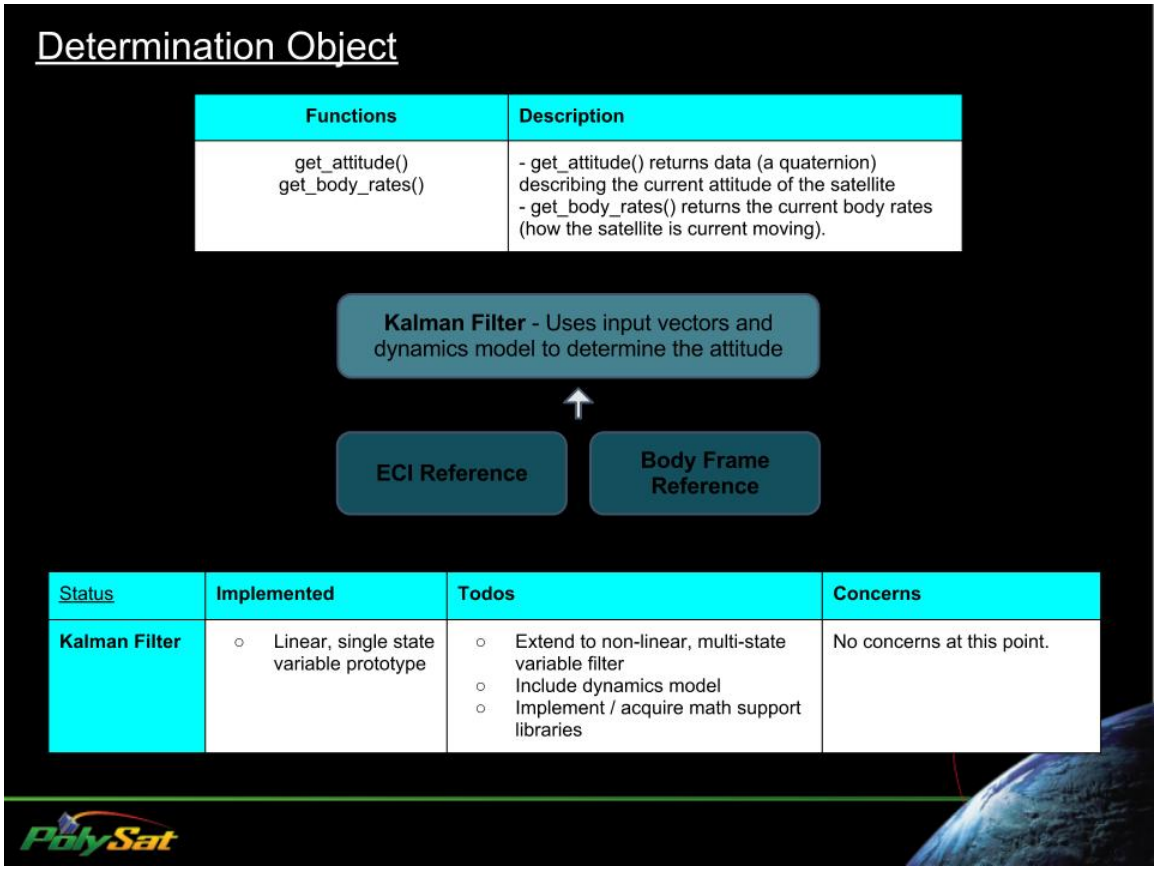


Figure 4: Determination Object

As shown in the figure, this Object will be responsible for getting the attitude as well as the body rates. The body rates will be incorporated into the dynamics model already. The reason to make them available is to give the control portion of the ADCS more knowledge on how to orientate the satellite. The middle, dark-green boxes break down the software with the 'Kalman Filter' box being a part of this Object while the other two are there just to give context.

Kalman Filter Design Overview

The Kalman Filter is within an overall ADCS system. This has been covered from the perspective of functional blocks. It takes a different form when looked at from a software perspective. As mentioned, the software design focuses on creating object like structures in C. Using this approach, the Kalman Filter code can be made module and tightly control access to important data. Another design criteria held as important was flexibility to future changes.

Modularity

Creating modular code requires the client to be as loosely tied to the specific implementation of a given component as possible. For instance, the Kalman Filter needs to have memory of its past state thereby making it necessary to store many variables that are never used outside the Kalman Filter. To create modular code, the task of holding these values should not include the

client as that would create a more tightly bound relationship. Also, the client should not handle values used internally to the Kalman Filter unless through explicit function calls to tightly control access to this internal data. To this end, a method of controlled initialization and struct casting has been used. When the client runs the initialization code for a Kalman Filter, it receives a 'KalmanFilter' structure that is actually a '__KalmanFilter' structure casted to a 'KalmanFilter.' The client uses a 'KalmanFitler' passing it to the provided functions which then use it as a '__KalmanFilter.' This allows one to have 'private' data in the structure that is only known about within the Kalman Filter libraries.

The structures:

```
// The struct used by the client
struct KalmanFilter {
  int (*iterate_kalman)(struct KalmanFilter *self, struct KalmanIn*,
                        struct KalmanOut*);
  int (*cleanup)(struct KalmanFilter *self);
};

// The struct used internally
struct __KalmanFilter {
  struct KalmanFilter filter;
  // Private variables
  struct KalmanSysVars sysVars;
  struct KalmanInternals internals;
  int numStateVars;
};
```

A stripped down client use case:

```
// The Kalman Filter pointer
struct KalmanFilter * kf;
// Inputs to the Kalman Filter algorithm
struct KalmanIn kIn;
// The Kalman Filter output
struct KalmanOut kOut;

// Leaving out the code that initializes the input

kf = init_kalman_filter(&sysVars, &initialGuess, numStateVars);

// Leaving out the code that initializes the Kalman Filter input

// Iterate the Kalman filter once
kf->iterate_kalman(kf, &kIn, &kOut);
```

Key snippets of the initialization:

```
struct KalmanFilter * init_kalman_filter(struct KalmanSysVars * sysVars,
                                         struct KalmanInternals * initialGuess,
```

```
                                int numStateVars)
{
        struct __KalmanFilter *new_filter;
        new_filter = (struct __KalmanFilter *) malloc(sizeof(struct __KalmanFilter));

        //Initialization code here

        return (struct KalmanFilter *) new_filter;
}
```

Key snippets of the iteration:

```
int iterate_kalman(struct KalmanFilter *self, struct KalmanIn *kalmanIn,
                struct KalmanOut * kalmanOut)
{
        struct __KalmanFilter *filter;

        if (!self)
                return 1;
        filter = (struct __KalmanFilter *)self;

        // Kalman Filter code here
}
```

Flexibility

As mentioned, this Kalman Filter is based off a design implemented by AERO students in Matlab. Although a working design has been implemented, it has yet to be finalized so flexibility in design changes is important to incorporate into the ported code. To this end, an extensive use of functions was used to create small modular blocks of code that can easily be changed and moved if needed. The high level functions are based around the generic functionality that applies to all Kalman Filters. Also, all the variables that are used to configure the Kalman Filter such as ‘R’ and ‘Q,’ are read in from a text file during initialization. Since the satellite has the ability to upload files, this provides the possibility of in-flight changes to the behavior of the filter. Along these lines, as a later addition to the current design, the ability to change these variables via the already implemented UDP packet command framework could be added.

Another design feature that will hopefully add flexibility to future design changes is the use of function pointers for the generic operations of the filter. For example, the filter structure has the following function pointer member:

```
// Function pointer for generic kalman filter functionality
int (*iterate_kalman)(struct KalmanFilter *self, struct KalmanIn*,
                struct KalmanOut*);
```

The API of this function should fit any specific implementation of a Kalman Filter. This gives one the ability to simply initialize this function callback differently to use a different specific implementation. Since there are a number of different flavors of a Kalman Filter, this may be a useful feature.

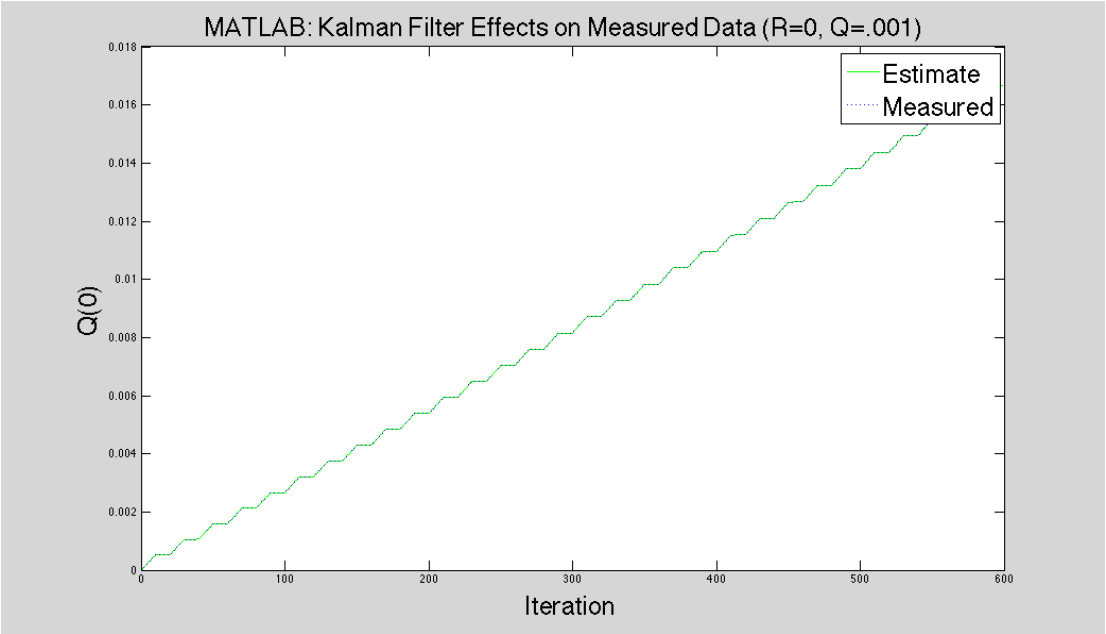
Results and Testing

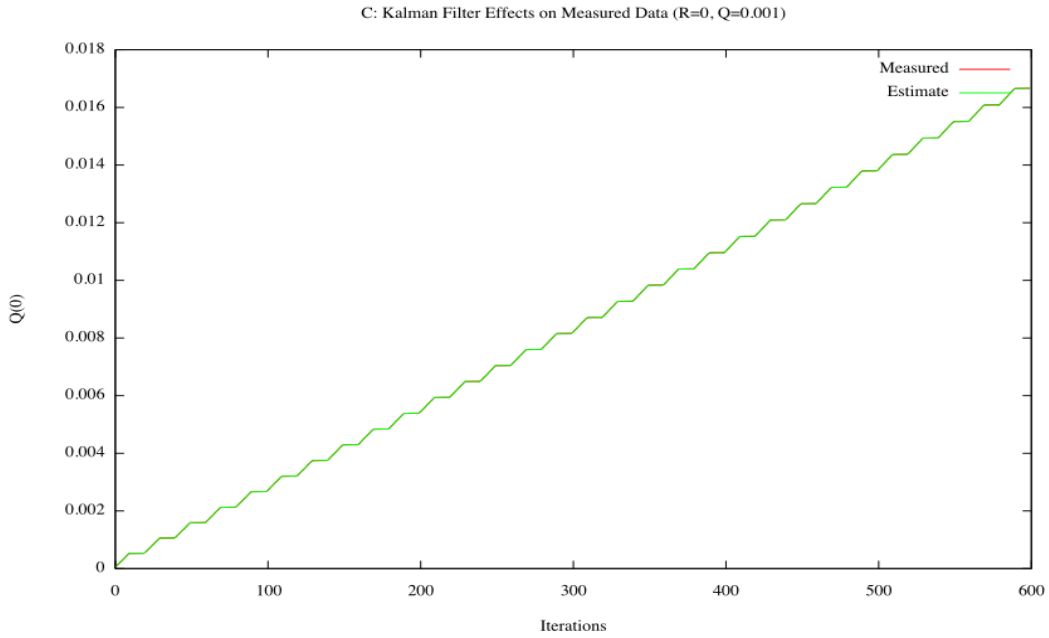
The primary verification of the Kalman Filter was its correspondence to the equivalent filter in Matlab. As mentioned, a Kalman Filter has various design variables that allow one to adjust its runtime behavior. These variables were adjusted to ensure correct functionality across a range of different settings. The two main variables that were adjusted were the covariance matrix of the process noise and the measurement noise - ‘Q’ and ‘R’ respectively.

Following is a series of pairs of Matlab filter and C filter graphs ran with the same design variables. The graphs show how the two filters match in terms of the estimated value relating to the measured value. Along with the graphs, 600 estimates were averaged for each configuration of the two filters giving the percent differences.

Graph Comparisons

Plot Pair 1: R=0, Q=.001

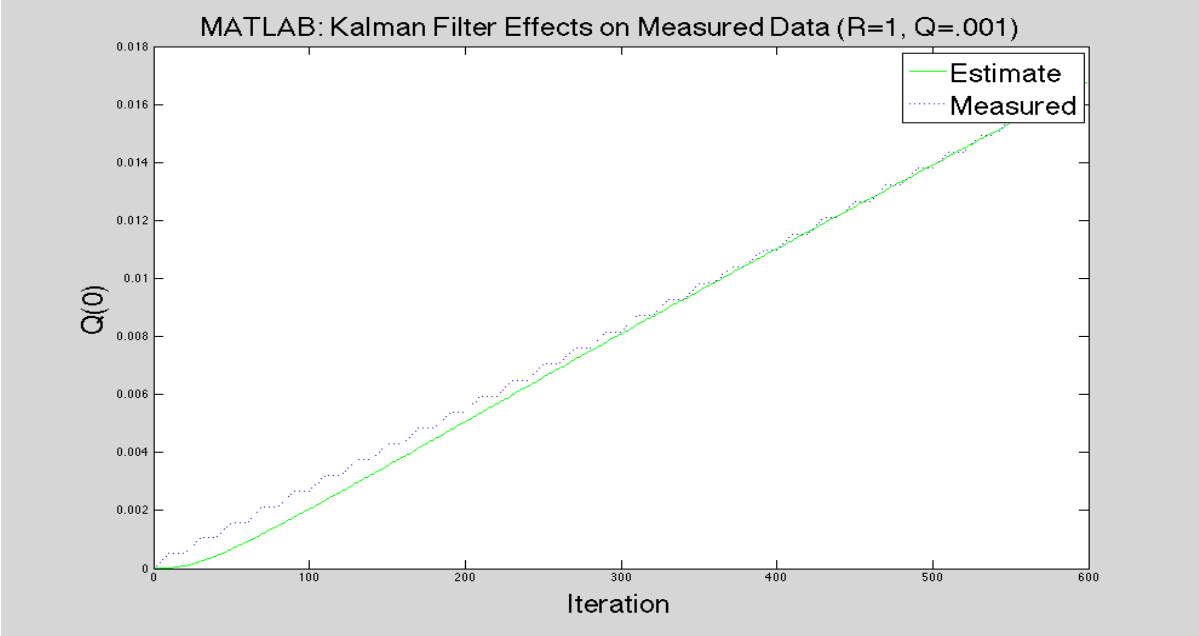


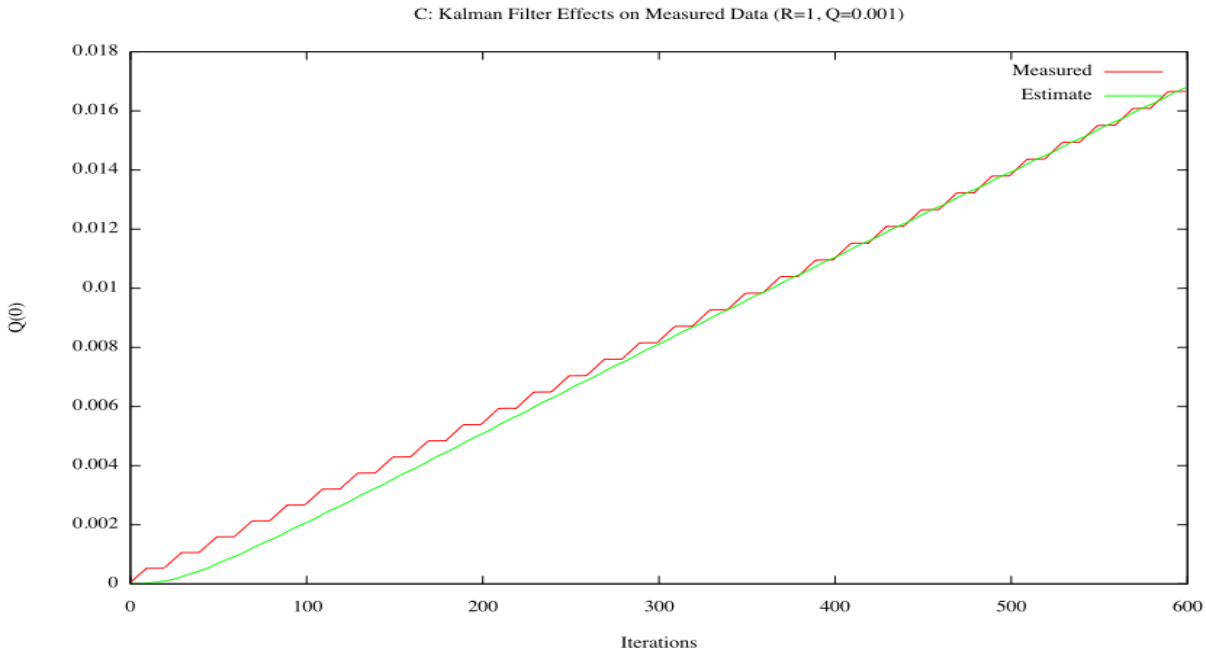


In the first pair, ‘R’ is zero. Since ‘R’ is the covariance matrix modeling the sensor noise, this

affects how much the filter factors in the measurement for the estimate. When ‘R’ is zero, the filter behaves as if the sensors were perfect, trusting them completely. This explains why the measured values and the estimate values in the above graphs are identical.

Plot Pair 2: R=1, Q=.001

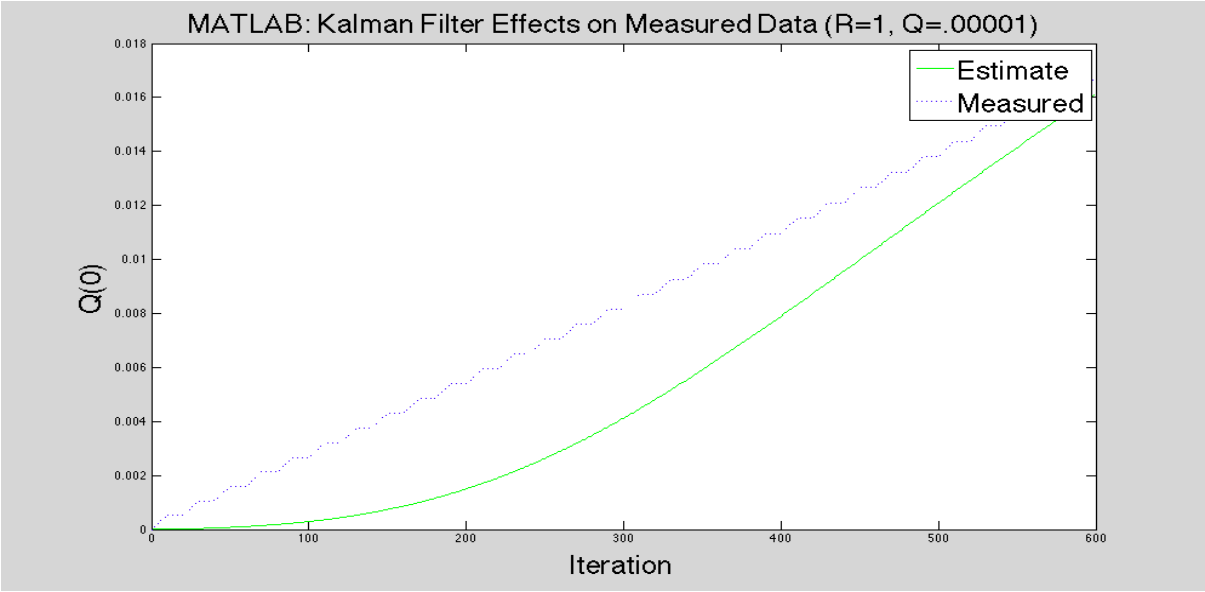


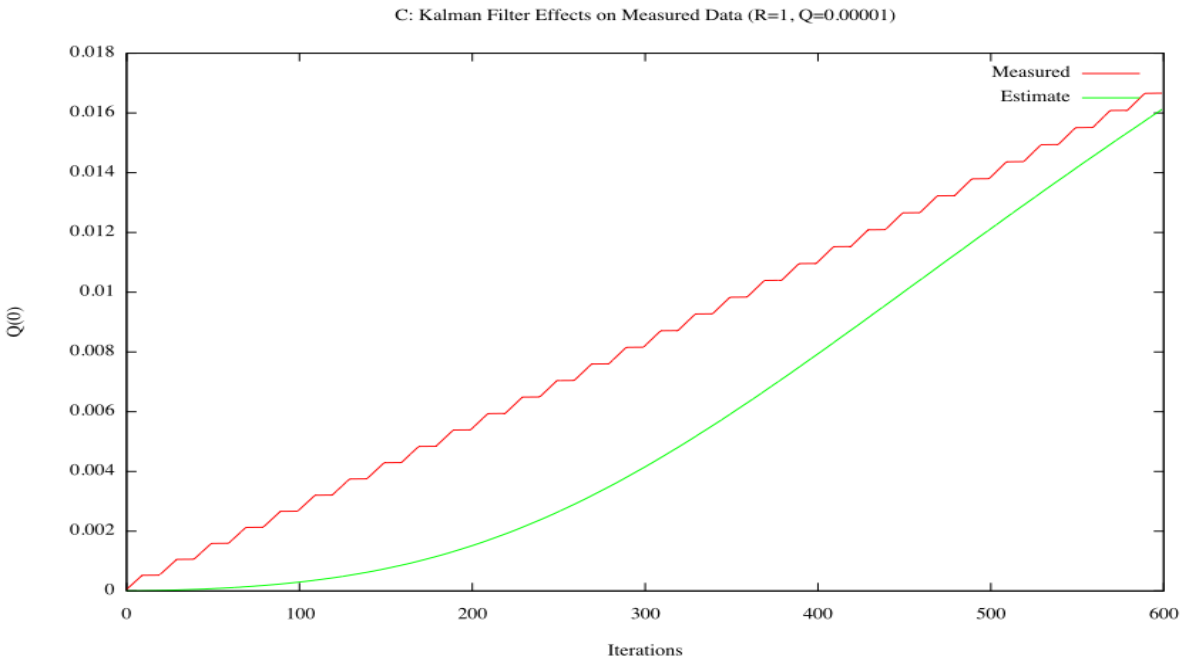


In the second pair, R is set to one giving the sensors noise. This is shown in how the estimate

does not conform to the measured values as it did in the first pair. Instead, it slowly moves towards the measured values, eventually outputting an somewhat averaged reading of them. Initially, the C code estimate did not converge to the measured value which proved to be an issue with the time steps involved per iteration. As shown above, both graphs now converge as they should.

Plot Pair 3: R=1, Q=.00001





This final pair is very similar to the 2nd pair but with a lower value of ‘Q.’ This gives the filter a

lower process noise. Since the process noise is similar to how much the filter trusts the prediction, a lower value makes the prediction stronger, making it take even longer to converge to the measured values. This is obvious since both the this and the 2nd pair of graphs went for 600 iterations yet this one did not converge. Even without converging, it still shows that both the MATLAB and the C filter behaved the same when the value of ‘Q’ was changed providing useful feedback.

Percent Differences

To calculate the percent difference, 600 iterations of each filter was averaged for each configuration. The iterations output four variables for the quaternion. Then the difference of the two divided by the MATLAB result was taken to get the percent difference.

Percent Differences

Q	**R**	**Q(0)**	**Q(1)**	**Q(2)**	**Q(3)**
.001	0	0.0011%	0.0012%	0.0004%	0.0002%
.001	1	0.0012%	0.0011%	0.0004%	0.0002%
.00001	1	0.0011%	0.001%	0.0014%	0.0000%

As shown above, the percent differences are very small and will not significantly affect the performance of the Kalman Filter.

Timing Analysis

An important aspect of the ported Kalman Filter to test is how it runs on the target architecture. The first check is compilation and execution where problems can arise. Thankfully, the target architecture and its already-developed software in this case supports all the major libraries and functionalities needed for the majority of C programs making simple execution a non-issue. Another important aspect is the runtime. In this case, the target machine does not have floating point hardware and instead simulates it which adds time. Since the Kalman Filter and the matrix library both use floating point, a timing analysis was an important test. However, the requirements in this area are thankfully loose as the Kalman Filter will not be expected to run in quick succession. The exact upper bounds on time is not precisely known but anything less than 100 milliseconds for an iteration is within the acceptable range.

For the analysis, the filter was iterated 600 times to get an average running time. Two different times were taken: The CPU time giving the time the process is actually running, and the wall time which times from the start of the execution to the end whether or not the process has been

swapped out or not. (Note that the precisions are different due to the Unix utilities used. For ‘CPU time,’ the ‘clock()’ function of the ‘time.h’ library was used giving millisecond precision. For the ‘wall time’ the ‘gettimeofday()’ function of the ‘sys/time.h’ library was used giving microsecond precision.) As shown below, both times were well within the acceptable range. The unexpected result of the CPU time being longer than the wall time can be explained by the greater precision of the wall time.

Timing Results

Time	**Iterations**	**Result**	**One Iteration Average**
CPU	600	900 ms	1.5 ms
Wall	600	899.833 ms	1.500 ms

Conclusion

The ported Kalman Filter has been shown to work accurately and efficiently enough to be used within the overall ADCS system. The percent differences of the resulting output between the MATLAB and C filter are small enough to ignore while the runtime on the target system is beyond what is needed. This is a good first step towards a final Kalman Filter. It is only a first step since, as mentioned, the AEROs did not provide a finalized design in time for the project. However, the design provided gives the general functionality needed for any Kalman Filter and is a good base for generating confidence in the target machines ability to effectively run any Kalman Filter needed. The implemented code should also make for changes in the design to be easily accommodated.

Works Cited

[1] Humphreys, Todd. Attitude Determination for Small Satellites with Modest Pointing Constraints. Utah State University, UT, 2003.
[2] "Geomagnetic Calculators, Maps, Models and Software." *National Geophysical Data Center*. N.p.. Web. 4 Dec 2012.

Credit for the coordinate system slides goes to Ryan Sellars